REPORT DOCUMENTATION PAGE

Form Approved OMB NO. 0704-0188

1. REPORT DATE (DD-MM-YYYY)	2. REPORT TYPE	3. DATES COVERED (From - To)
30-06-2016	Final Report	5-Aug-2013 - 4-Apr-2016

4. TITLE AND SUBTITLE
Final Report: FAIL-SAFE: Fault Aware IntelLigent SoftwAre For Exascale

5a. CONTRACT NUMBER
W911NF-13-1-0219

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHORS
Robert F. Lucas

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES AND ADDRESSES
University of Southern California
Department of Contracts and Grants
3720 South Flower Street
Los Angeles, CA 90089 -0701

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS (ES)
U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSOR/MONITOR'S ACRONYM(S)
ARO

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
63295-CS-ACI.9

12. DISTRIBUTION AVAILIBILITY STATEMENT
Approved for Public Release; Distribution Unlimited

13. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not contrued as an official Department of the Army position, policy or decision, unless so designated by other documentation.

14. ABSTRACT
The University of Southern California (USC), the Lawrence Livermore National Laboratory (LLNL), and the Jet Propulsion Laboratory (JPL) believe that a new generation of dependable applications must be developed to successfully exploit this next generation of technology. Such applications and the systems they run on must be introspective and adaptive, actively searching for errors in their program state with hardware mechanisms and new software techniques. Towards this end,we have developed and demonstrating the technology to enable adaptive, application-oriented control of fault tolerance, for a set of scientific applications on a workstation-class system by

15. SUBJECT TERMS
Resilience, Exascale, Software Directives, Program Analysis and Transformations, Introspection,

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	15. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Robert Lucas
UU	UU	UU	UU		19b. TELEPHONE NUMBER 310-508-4118

Standard Form 298 (Rev 8/98)
Prescribed by ANSI Std. Z39.18

Report Title

Final Report: FAIL-SAFE: Fault Aware IntelLigent SoftwAre For Exascale

ABSTRACT

The University of Southern California (USC), the Lawrence Livermore National Laboratory (LLNL), and the Jet Propulsion Laboratory (JPL) believe that a new generation of dependable applications must be developed to successfully exploit this next generation of technology. Such applications and the systems they run on must be introspective and adaptive, actively searching for errors in their program state with hardware mechanisms and new software techniques. Towards this end,we have developed and demonstrating the technology to enable adaptive, application-oriented control of fault tolerance, for a set of scientific applications on a workstation-class system by injecting memory faults and observing the survivability of the applications.

We have defined an assertion language that provides programmer with a convenient interface to specific the resilient characteristics of applications and have implemented a limited set of these assertions as source-to-source transformations in the ROSE-compiler infrastructure. The outcomes of this research provide a model for the vendors of Defense systems, and a prototype capability should the vendors chose not to bring such technology to market. The increased application resilience resulting from this research will lead to faster completion of Defense applications, and thus substantial energy savings as well as increased mission assurance.

Enter List of papers submitted or published that acknowledge ARO support from the start of the project to the date of this printing. List the papers, including journal references, in the following categories:

(a) Papers published in peer-reviewed journals (N/A for none)

Received Paper

TOTAL:

Number of Papers published in peer-reviewed journals:

(b) Papers published in non-peer-reviewed journals (N/A for none)

Received Paper

TOTAL:

Number of Papers published in non peer-reviewed journals:

(c) Presentations

1. FAIL-SAFE: Fault Aware IntelLigent SoftwAre For Exascale, Bob Lucas, Presentation at the Advanced Computing Initiative Kickoff, June 2013

2.FAIL-SAFE: Fault Aware IntelLigent SoftwAre For Exascale, Bob Lucas, Poster Presentation PI Meeting, Feb, 2014

3.FAIL-SAFE: Fault Aware IntelLigent SoftwAre For Exascale, Bob Lucas, Poster Presentation PI Meeting, July, 2014

4. The FAIL-SAFE Assertion Language, Hans P. Zima, Jacqueline Chame, Pedro C. Diniz, Robert F. Lucas, Presentation at the Advanced Computing Initiative Meeting, Marina del Rey, Sep. 2014.

5. Spacecraft Health Inference Engine (SHINE): A Tool for Building and Deploying Real-time Rule-based Reasoning Systems for Detection, Diagnostics, Prognostics, Recovery, and Network Management, Mark James, 2015.

6. S. Hukerikar, P. Diniz, and R. Lucas, “Resilience for Exascale HPC Systems, A Programming Model Approach”, SIAM Conf. on Parallel Processing for Scientific Computing, Feb. 2014, Portland, OR, USA

Number of Presentations: 6.00

Non Peer-Reviewed Conference Proceeding publications (other than abstracts):

Received		Paper
06/29/2016	5.00	. Enabling Application Resilience through Programming Model based Fault Amelioration, Proc. of the IEEE High Performance Embedded Computing Conference. 10-SEP-57, Waltham, MA. : ,
06/29/2016	6.00	. Pragma-controlled Source-to-Source Code Transformations for Robust Application Execution, 9th Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids 2016. 23-AUG-73, Grenoble, France. : ,
TOTAL:	**2**	

Number of Non Peer-Reviewed Conference Proceeding publications (other than abstracts):

Peer-Reviewed Conference Proceeding publications (other than abstracts):

Received	Paper	
06/30/2016	2.00	Saurabh Hukerikar, Keita Teranishi, Pedro Diniz, Robert Lucas. Opportunistic Application-level Fault Detection through Adaptive Redundant Multithreading, HPCS 2014. 24-JUL-14, Bologna, Italy. : ,
06/30/2016	3.00	Saurabh Hukerikar, Keita Teranishi, Pedro Diniz, Robert Lucas. An Evaluation of Lazy Fault Detection based on Adaptive Redundant Multithreading, HPEC 2014. 11-SEP-14, Waltham, MA. : ,
TOTAL:	**2**	

Number of Peer-Reviewed Conference Proceeding publications (other than abstracts):

(d) Manuscripts

Received	Paper
TOTAL:	

Number of Manuscripts:

Books

Received	Book
TOTAL:	

Received	Book Chapter

TOTAL:

Patents Submitted

Patents Awarded

Awards

Graduate Students

NAME	PERCENT_SUPPORTED	Discipline
Saurabh Hukerikar	1.00	
FTE Equivalent:	**1.00**	
Total Number:	**1**	

Names of Post Doctorates

NAME	PERCENT_SUPPORTED
FTE Equivalent:	
Total Number:	

Names of Faculty Supported

NAME	PERCENT_SUPPORTED
FTE Equivalent:	
Total Number:	

Names of Under Graduate students supported

NAME	PERCENT_SUPPORTED
FTE Equivalent:	
Total Number:	

Student Metrics

This section only applies to graduating undergraduates supported by this agreement in this reporting period

The number of undergraduates funded by this agreement who graduated during this period: 0.00

The number of undergraduates funded by this agreement who graduated during this period with a degree in science, mathematics, engineering, or technology fields:..... 0.00

The number of undergraduates funded by your agreement who graduated during this period and will continue to pursue a graduate or Ph.D. degree in science, mathematics, engineering, or technology fields:..... 0.00

Number of graduating undergraduates who achieved a 3.5 GPA to 4.0 (4.0 max scale):..... 0.00

Number of graduating undergraduates funded by a DoD funded Center of Excellence grant for Education, Research and Engineering:..... 0.00

The number of undergraduates funded by your agreement who graduated during this period and intend to work for the Department of Defense 0.00

The number of undergraduates funded by your agreement who graduated during this period and will receive scholarships or fellowships for further studies in science, mathematics, engineering or technology fields:..... 0.00

Names of Personnel receiving masters degrees

NAME

Total Number:

Names of personnel receiving PHDs

NAME
Saurabh Hukerikar
Total Number: **1**

Names of other research staff

NAME PERCENT_SUPPORTED

FTE Equivalent:
Total Number:

Sub Contractors (DD882)

1 a. Lawrence Livermore National Laboratory (LLNL) 1 b. 7000 East Ave

Livermore CA 945509698

Sub Contractor Numbers (c): L-15140
Patent Clause Number (d-1):
Patent Date (d-2):
Work Description (e): Extended the ROSE compiler to parse ISI's resilience directives and assertions.
Sub Contract Award Date (f-1): 12/14/14 12:00AM
Sub Contract Est Completion Date(f-2): 2/29/16 12:00AM

---

1 a. Lawrence Livermore National Laboratory (LLNL) 1 b. P. O. Box 808

Livermore CA 945500622

Sub Contractor Numbers (c): L-15140
Patent Clause Number (d-1):
Patent Date (d-2):
Work Description (e): Extended the ROSE compiler to parse ISI's resilience directives and assertions.
Sub Contract Award Date (f-1): 12/14/14 12:00AM
Sub Contract Est Completion Date(f-2): 2/29/16 12:00AM

---

1 a. NASA Jet Propulsion Laboratory 1 b. Quantum Sciences and Technology (

4800 Oak Grove Drive

Pasadana CA 91109

Sub Contractor Numbers (c): 10142984
Patent Clause Number (d-1):
Patent Date (d-2):
Work Description (e): Used SHINE (Spacecraft Heath INference Engine) to generate rules for ISI's resilience
Sub Contract Award Date (f-1): 12/14/14 12:00AM
Sub Contract Est Completion Date(f-2): 2/29/16 12:00AM

---

1 a. NASA Jet Propulsion Laboratory 1 b. Quantum Sciences and Technology (

4800 Oak Grove Drive

Pasadana CA 91109

Sub Contractor Numbers (c): 10142984
Patent Clause Number (d-1):
Patent Date (d-2):
Work Description (e): Used SHINE (Spacecraft Heath INference Engine) to generate rules for ISI's resilience
Sub Contract Award Date (f-1): 12/14/14 12:00AM
Sub Contract Est Completion Date(f-2): 2/29/16 12:00AM

Inventions (DD882)

Scientific Progress

Prior to FAIL-SAFE, we had already demonstrated that a knowledgeable user can assert what regions of a program's state space can tolerate errors, and that these programs can continue to correct solutions. To broaden the impact of this research, we also needed to be able to ameliorate errors, with minimal overhead. Today, this is largely done by check-pointing state, and then rolling back when errors are detected, and restarting the code. We demonstrated that one can extend a standard programming API to allow a knowledgeable user to be able to provide alternative repair strategies that will reduce the frequency of check-pointing and restarting, thus saving time and energy. What these are, and how broadly applicable they will be, remains an open research question.

Today's standard model of computation, embodied in familiar programming languages, assumes that the underlying computer runs correctly. This model is generally accepted, except in safety critical systems like flight controls. In the near future, this will no longer be true due to the continued scaling of VSLI. In addition it will be prohibitively expensive to enforce total operational correctness with error correction using redundancy. The results of the research carried out in this project will be software that exploits human knowledge of what faults are significant, and what are not, to reduce the overhead of maintaining the illusion of perfect computing systems. This will save time and energy for large-scale Defense computations.

Technology Transfer

We have distributed drafts of our assertion language to two DOD ACS program investigators, Prof. Vivek Sarkar at Rice University and Dr. David Bernholdt at Oak Ridge National Laboratory. Per the direction of John Daly, we also shared it with Erik deBenedictis of Sandia National Laboratory and Marti Bancroft, a contractor for another DOD organization. We demonstrated resilient execution of the HPCS Random Access benchmark at the DOD ACS workshop, July 17, 2014. We have also published in the scientific literature, as enumerated below. Finally, we have installed at NSA R3 an up-to-date release of the ROSE Compiler Infrastructure and an implementation of a selected set of the ROSE-based source-to-source transformations that support the resilience directives developed as part of the FAI-SAFE language at a computer cluster at the direction of the Government.

FAIL SAFE: Fault Aware IntelLigent SoftwAre For Exascale Final Report

June 13, 2016

Award number W911NF-13-1-0219

Robert F. Lucas
Information Sciences Institute
University of Southern California
(310)448-9449
rflucas@isi.edu

Introduction

By the end of this decade, the Department of Defense will be deploying large numbers of massively parallel systems to address a broad set of problems ranging from mission critical challenges such as cryptanalysis, image processing, decision support, and weather forecasting to fundamental research questions in science and technology. These high performance-computing systems will be constructed from exascale technology. As such, they will be composed of devices less reliable as those used today, and faults will become the norm, not the exception. This will pose significant problems for Defense users, who for half a century have enjoyed an execution model that largely relied on correct behavior by the underlying computing system.

The University of Southern California (USC), the Lawrence Livermore National Laboratory (LLNL), and the Jet Propulsion Laboratory (JPL) believe that a new generation of dependable applications must be developed to successfully exploit this next generation of technology. Such applications and the systems they run on must be introspective and adaptive, actively searching for errors in their program state with hardware mechanisms and new software techniques. Towards this end, the Army Research Office (ARO) funded us via contract number W911NF-13-1-0219 to perform research with the goal of developing and demonstrating the technology to enable adaptive, application-oriented control of fault tolerance. Our initial plan was to extend the ROSE compiler, the LLVM compiler, and the SHINE introspection engine so that faults injected into a resilient application's state will be detected and dealt with, whether by ignoring them, correcting them if possible, or reverting to an earlier checkpoint when necessary. A reduction in scope from the proposal prevented us from pursuing the work with LLVM.

The report provides an overview of the results of this research. Further details can be found in the publications listed below, the PhD thesis of Dr. Saurabh Hukerikar, and software that was delivered to NSA's R3. The remainder of this report is organized along the lines requested by ARO for previous reports.

Objective

In the FAIL-SAFE project, we extended the familiar C programming language to allow software developers to express their knowledge of the fault tolerance of their applications. The approach pursued, uses a high-level annotation language for expressing user knowledge about runtime correctness conditions, error tolerance, and specific methods for enhancing reliability. As a user-

controlled approach, this approach avoids the performance and/or energy penalties associated with the blind application of traditional redundancy methods such as hardware/software Triple-Modular Redundancy (TMR). We extended our initial resilience framework along with the ROSE compiler and the SHINE introspection engine so that faults injected into a resilient application's state can be detected and dealt with, whether by ignoring them, correcting them if possible, or reverting to an earlier checkpoint when necessary. The results of this research provide a model for the vendors of Defense systems, and a prototype capability should the vendors chose not to bring such technology to market. The increased application resilience resulting from this research will lead to faster completion of Defense applications, and thus substantial energy savings as well as increased mission assurance.

Approach

At the beginning of the FAIL-SAFE project, USC has an existing fault tolerance test bed, constructed with prior research funding that demonstrates that some uncorrectable errors can be ignored and applications still continue to correct solutions. The FAIL-SAFE project created a major generalization of this system by (1) developing a high-level annotation language; (2) automating the translation of directives in this high-level annotation language as source-to-source code transformations in the DoE-funded and supported ROSE compiler infrastructure (3) designing an interface between the application and an introspection framework for resilience (IFR) based on the inference engine SHINE; (4) using the ROSE compiler to translate annotations into reasoning rules for the IFR; and (5) designing a Knowledge/Experience Database, which will store knowledge about dynamic program behavior and resilience determined by the IFR to be leveraged in subsequent program development cycles. We implemented this this technology as open-source software so that Defense users have access to it.

Scientific Barriers

Prior to FAIL-SAFE, we had already demonstrated that a knowledgeable user can assert what regions of a program's state space can tolerate errors, and that these programs can continue to correct solutions. To broaden the impact of this research, we also needed to be able to ameliorate errors, with minimal overhead. Today, this is largely done by check-pointing state, and then rolling back when errors are detected, and restarting the code. We demonstrated that one can extend a standard programming API to allow a knowledgeable user to be able to provide alternative repair strategies that will reduce the frequency of check-pointing and restarting, thus saving time and energy. What these are, and how broadly applicable they will be, remains an open research question.

Significance

Today's standard model of computation, embodied in familiar programming languages, assumes that the underlying computer runs correctly. This model is generally accepted, except in safety critical systems like flight controls. In the near future, this will no longer be true due to the continued scaling of VSLI. In addition it will be prohibitively expensive to enforce total operational correctness with error correction using redundancy. The results of the research carried out in this project will be software that exploits human knowledge of what faults are significant, and what are not, to reduce the overhead of maintaining the illusion of perfect computing systems. This will save time and energy for large-scale Defense computations.

Accomplishments

The FAIL-SAFE project was initially funded in August of 2013. Because we partnered with National laboratories (LLNL and JPL), in took months to get them on contract. In fact, due to inconsistencies between the FAR and the Space Act, ARO had to intervene and waive some clauses. So, in a very real sense the FAIL-SAFE team only came together in early 2014.

As first reported in August 2014, the FAIL-SAFE team drafted an assertion language that allows users to communicate fault tolerance knowledge to a compiler and underlying computing systems. LLNL has extended ROSE to parse our initial assertion language. The assertion language is itself a work in progress, and this was an ongoing process throughout the remainder of the project. The most recent version, Version 8.0, was distributed in August of 2015, after we began collaborating with Dr. Erik DeBenedictis from Sandia National Laboratories (SNL) at the direction of the government. It provided the assertion language support for key concepts outlined in Erik's white paper entitled "Managing the End of Moore's Law". It is included as an appendix to this report. JPL also delivered to USC a first prototype of a SHINE-generated Introspection Framework for Resilience (IFR). This has been tested in the USC resilience test bed, running on USC's HPC cluster and was demonstrated at the July 2014 DOD ACS program workshop. Finally, LLNL implemented aspects of the assertion language in ROSE, software that has been delivered to NSA R3. These accomplishments are discussed in more detail below.

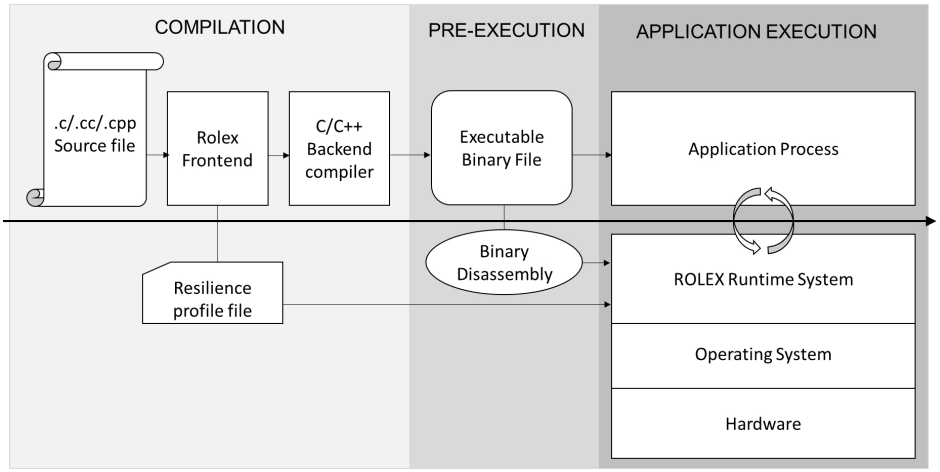

Figure 1. Application compilation and fault-injection infrastructure.

Accomplishments: Fault Detection and Amelioration

USC developed a fault-injection and amelioration analysis infrastructure, depicted in Figure 1 above. Using this infrastructure, we developed a set of computational kernels to be analyzed for robustness and refined the type of failures observed. Figure 2 below depicts the impact of faults randomly injected into two computational kernels. The injection rate varies from once every fifteen minutes, to once per minute. The outcomes are: faults are detected and corrected; benign faults where the application succeeds even though faults are silent; undetected faults leading to incorrect computational outcomes; and application crashes. The Graph 500 breadth-first-search algorithm contains several pointer-related computations to traverse the graph edges. Therefore, almost 50% of the execution runs can detect and correct the corruptions in the pointer arithmetic. However, since the other parts of the computational environment as well the graph vertex data elements

contain no error management knowledge, the application fails more often at very high fault rates. The AMG code is naturally resilient to errors. Most of its memory is allocated to the intermediate solution grids at each level in the V-cycle, and as an iterative algorithm, it can often reach a solution in spite of errors. The pointer arithmetic is the most sensitive to silent corruptions and the robust qualifiers create redundant copies which allow detection and correction for errors injected in these variables.

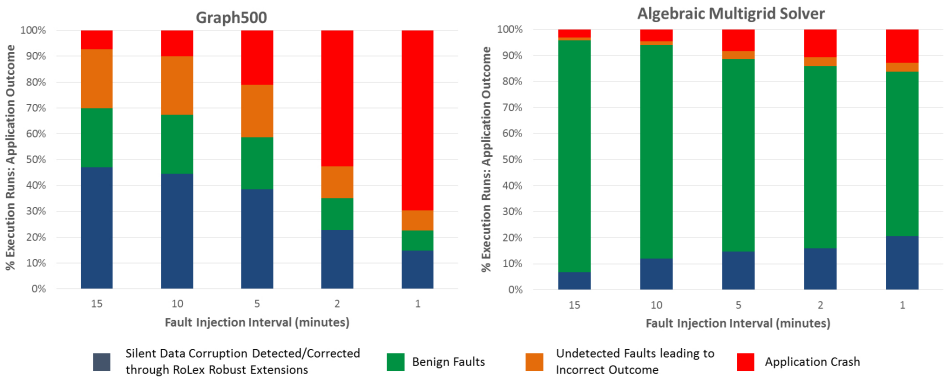

Figure 2. Evaluation of application kernel robustness for various fault-injection rates.

At direction of the government, we also implemented simple amelioration strategies to two Conjugate Gradient implementations (a traditional CG and a Self-Stabilizing CG) as well as a algorithmic-based fault-tolerance (ABFT) strategy for the popular matrix-matrix multiplication DGEMM kernel. In the case of DGEMM there are column- and row-wise checksums that allow the implementation to recover the correct matrix element value in the presence of an error. For the CG kernels the implemented amelioration procedures were respectively a roll-forward and a roll-backwards to the previous iteration of the algorithm and using selected saved data.

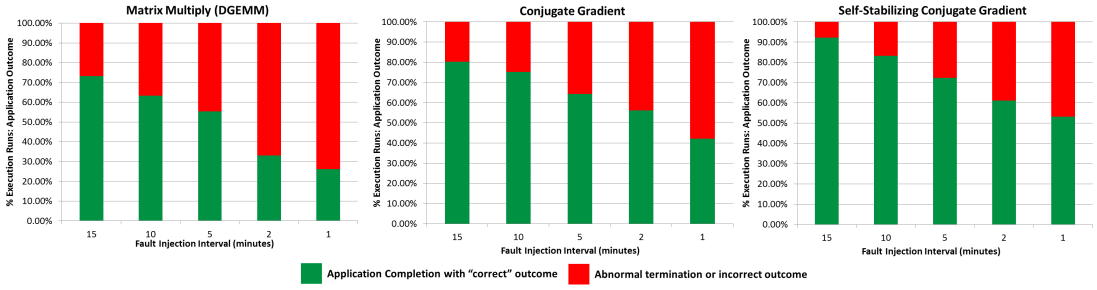

Figure 3. Evaluation of Simple Amelioration strategies for DGEMM (ABFT) and Conjugate Gradient (roll-forward and roll-backwards).

Figure 3 above summarizes the results of these experiments for various fault injection rates. For the DGEMM code, the checksum-based amelioration is applicable for only the static data, i.e. the operand matrices which are initialized at the beginning and whose values do not change throughout the execution. A total of 75% of all executions converge correctly for a rate that injects an error every 5 minutes but only 27% complete correctly at the accelerated rate of an error per minute. For the CG computation, we leverage the iterative nature of the algorithm that allows the execution to often overcome errors on the solution vector while the operand matrices are protected using checksums.

Many faults are only detectable by the appearance of erroneous data in program state. N-modular redundancy is the traditional mechanism for detecting these, and the overhead exceeds N as there are synchronization and comparisons added to implement it, not just multiple copies of the computation. The FAIL-SAFE project explored an adaptive redundant multithreading variation of this well know technique. The compiler replicates code blocks identified by the programmer that can serve as spheres of replication, where values can be checked before errors propagate beyond the boundaries. An introspective runtime system can enable or disable redundancy for any of these code spheres based on the fault rates it is observing. As shown below, in figure 4, this reduces the overhead of using redundancy to test for the presence of errors, relative to process pairing.

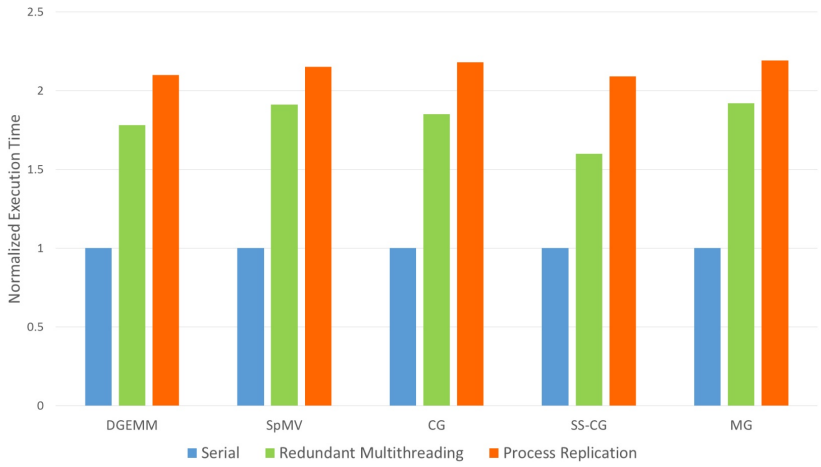

Figure 4: Adaptive redundant multithreading reduces overhead relative to full process replication.

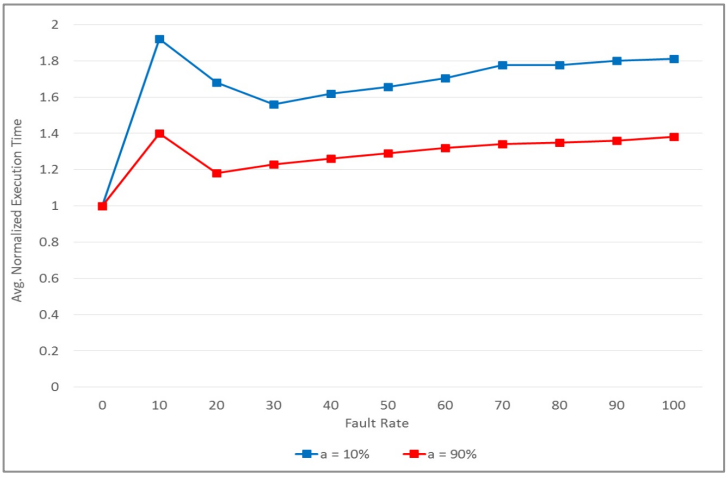

Figure 5: Performance overhead of fault aware thread assignment for modeling accuracy rates of 10 and 90%. Note that this quickly plateaus once the faulty cores have been isolated.

Another tool that an introspective system could have at its disposal is the ability to migrate tasks away from faulty cores. Faults in systems are correlated in both time and place. For example, studies have shown that 5% of locations account for 95% of faults. One can attempt to anticipate the occurrence of future faults based on historical data. Figure 5 above illustrates the overhead resulting from anticipating faults and migrating tasks. Note that the first ten or so errors injected

have a significant deleterious impact on performance, but that it quickly stabilizes as the fault rate increases. This is because the higher fault rates provide more data with which better predictions can be made, and faulty cores isolated.

Accomplishments: ROSE Implementation of the FAIL-SAFE Assertion Language

As part of the FAIL-SAFE project, LLNL defined a set of language annotations in the form of C/C++ pragmas to meet the requirements defined by the draft FAIL-SAFE Assertion Language. These were realized using LLNL's ROSE compiler transformation front-end, a tool whose ongoing development at LLNL made them a critical member of the FAIL-SAFE team. The FAIL-SAFE assertion language gives a set of abstract syntax and grammar rules to represent predicates and directives. Predicates are further categorized into status predicates and data predicates. Status predicates are associated with a point of execution related to a statement and specify a condition that needs to be valid whenever that point is reached during program execution. Data predicates are associated with objects generated in the context of a variable, type, or class declaration and specify a condition that must hold for these data throughout their lifetime. Directives include mostly tolerance and redundancy directives. A tolerance directive is to express tolerance for certain classes of errors occurring during the execution of the program, such as arithmetic or SECDED (single-error correction double-error detection) errors. Redundancy management directives are mainly used in high-reliability sections for the enforcement of hard correctness. They provide a set of methods based on the redundancy of code or data.

A concrete set of language annotations for actual programming languages are needed to instantiate the assertion language and enable programmers to use our assertions. Our design follows the pragma convention of OpenMP specification for easier understanding and compiler implementation. Both C/C++ and Fortran programs can also be uniformly handled using OpenMP-style pragmas.

Each directive starts with #pragma failsafe . Directives are case-sensitive. The general form is

#pragma failsafe directive-name [clause [[,] clause] ....] new-line

A FailSafe executable directive applies to at most one succeeding statement, which must be a structured block. Some representative example pragmas we define are depicted below:

```
// Status Predicates
...
x=f0(y);
#pragma failsafe status assert (x < U) error (ET3) recover (R3,x,y,U)
L1: z=g(x);
...

// Another status predicate
...
LOOP1: while cexpr
{
/* while-body */
#pragma failsafe status assert (x < y) error (...)
}
```

```
...
// Data predicates
// form 1: context based
...
int ncycles;
#pragma failsafe data assert (0 <= ncycles <= maxcycles)

// form 2:  using region reference in(R1)
...
int ncycles;
...
#pragma failsafe data assert (0 <= ncycles <= maxcycles) in (ncyles)

//form 3: using wild card keyword: allvars

int ncycles, mcycles;
#pragma failsafe data assert (0 <= allvars <= maxcycles)
```

Using the ROSE compiler infrastructure, we have also implemented the parsing support for the set of C/C++ pragmas we defined, including those specifying assertion region, status predicate, data predicate, violation types and so on. The implementation is based on the extensions to ROSE to parse input programs annotated with our pragmas and store the information as persistent attributes attached to ROSE's AST (abstract syntax tree). As a result, we now have a proper internal representation of programs using assertion language annotations.

These transformations include the translation of an important subset of directives and constructs defined in the FAIL-SAFE language and were validated early using the USC's tested-bed albeit in a semi-manual fashion. The automation of these directives is completed but there are still imitations about the data types and compiler analysis supported in the current software distribution. The software has been tested and delivered at the request of the Government on a computer cluster located at the University of Maryland and under the supervision of Dr. Simon Tyler.

Accomplishments: SHINE Component of the FAIL-SAFE IFR

The FAIL-SAFE IFR works by monitoring the run-time execution of an application and the system it runs on, and manages their responses to errors. Whenever an error occurred, its location, type, and severity, will all be communicated to the IFR which will analyze errors and subsystem failures, and provide feedback to the application, the operating system, and ultimately the developers and users of the application. The role of the IFR can be understood as encapsulating functionality related to the monitoring and analysis of special events that occur during the execution of the application, and reasoning about related problems and recovery strategies. For example, the IFR may react to failing correctness assertions, monitor the cache misses and performance characteristics of program sections, and reason about the frequency of errors over time and their correlation. In case of severe errors it may negotiate with the operating system about an appropriate recovery strategy.

At the heart of the IFR is the Spacecraft Health Inference Engine (SHINE) and a IFR knowledge base. In principle, this capability can be also applied to areas beyond fault tolerance, such as performance tuning, energy and power management, behavior analysis, and intrusion detection. SHINE was selected because it was specifically designed to monitor a highly instrumented system and react to

conditions in real-time. SHINE is capable of executing 100,000,000+ inferences per second where the second best inference engine at the time (CLIPS) only executed 40,000 rules per second. In order to actively monitor applications as they are running and to additionally react to their behaviors, the CLIPS engine was nowhere near fast enough. JPL has extensive experience using SHINE, making it a uniquely valuable team member for the FAIL-SAFE project.

SHINE is intended for those areas of inference where speed, portability, and reuse are of critical importance. Such areas have historically included spacecraft monitoring, control and health, telecommunication analysis, medical analysis, finical and stock market analysis, fraud detection (e.g. banking and credit cards), robotics or basically any area where rapid and immediate response to high-speed and rapidly changing data is required.

SHINE was originally designed to be embedded in single-threaded applications where the interruption from the inference cycle would only occur at task rescheduling points, where the complete system state was saved by the operating system and only a single SHINE was executing at any given point. It was not intended to be an all-encompassing expert system for mutually cooperating mini SHINEs all running simultaneously. Because of this assumption, all the rules and states of a knowledge base are reduced to a data flow graph with all possible dependency paths calculated in advanced. However, the FAIL-SAFE IFR required a far most robust control structure where many SHINEs were all executing in true parallelism and sharing results between them. This caused the state predictions by the compiler to be voided and incorrect inferences to be made.

SHINE introduces a novel paradigm for knowledge visualization and ultra-fast inference that goes well beyond traditional forward and backward chaining methodology. A sophisticated mathematical transformation based on graph-theoretic Data Flow analysis is introduced, that reduces the complexity of conflict-resolution during the match cycle from $O(n^2)$ to $O(n)$ for many kinds of inference operations. This transformation executes compiled SHINE knowledge bases at an excess of 33,000,000 rules per second on flight hardware and over 220,000,000 rules per second on a standard 3 GHz desktop PC.

A Data Flow program consists of data (1), which run the program, operations (2), which are activated when data is sent to them, and finally the results (3), which is what the program will return when completed. When data reaches a procedure it activates or "fires off" that procedure. Originally SHINE only contained what is called static firing, which meant a fairly broad set of graph optimizations could be applied to globally optimize the flow graph. To support the FAIL-SAFE IFR domain, we fundamentally changed the SHINE compiler to include two different ways to perform this "firing": (1) Static Data Flow: A procedure will begin when a piece of data is located at every input edge and no data is present at any output edge. Only one piece of data can reside on each edge and (2) Dynamic Data Flow: Each piece of data has some way of identify which other data it belongs with, such as a color, and when all the input edges contain data of the same type, the procedure will begin. Any input or output edge can handle multiple pieces of data.

A high priority goal for FAIL-SAFE was to avoid introducing an overall performance loss to SHINE by including dynamic firing so we introduced an additional analysis phase to only isolate those portions of a rule set that needed dynamic firing and leave the rest the same. Thus the dynamic firing portions are compiled with a thread-safe set of optimization rules that guaranteed valid state transitions between co-routine activated instances of SHINE.

Overall, we found Data Flow to be a very powerful method of parallel programming or representing knowledge that is executed in pseudo parallelism; however it is very difficult to write programs in a

Data Flow environment. Un-optimized Data Flow programs require a lot of storage and during execution there are scheduling problems, which must be controlled by some means of hardware or a software executive. As a result, very few machines have been developed for Data Flow and the chances are that very few will be developed for commercial use in the future since Data flow is expensive. However, Data Flow is an excellent means of representing information that can be executed in pseudo parallelism, optimized, and then mapped to a traditional architecture for execution. Obviously, many of the speed advantages of a true hardware Data Flow computer cannot be realized when executed on a serial architecture but it still offers an excellent frame work to represent parallel algorithms and an efficient means to execute them over traditional methods

Collaborations and Leveraged Funding

We have shared out draft assertion language with colleagues at Prof. Sarkar (Rice University), Dr. David Bernholdt (Oak Ridge National Laboratory), Dr. Erik Debenedictis (Sandia National Laboratory), and Marti Bancroft (DOD contractor). Saurabh Hukerikar also spent two summers as an intern at Sandia National Laboratory, and collaborated with Robert Clay.

At the direction of the Government, we have also explored with Dr. Benedictis and other of USC's research staff the trade-off between sub-threshold voltage computing in the context of computer architecture digital circuits and resilience for increased power efficiency. From this interaction resulted various internal discussions for potential future actionable items in this area of research.

While, this research has focused exclusively on software resilience, the USC component of this work builds on earlier funding form the Semiconductor Research Corporation (SRC), NSF, DARPA MTO (via SRC), and DOE ASCR's SciDAC-3 Institute for Sustained Performance, Energy, and Resilience (SUPER). Both ROSE and SHINE have long histories of support and use from DOE and NASA, respectively.

Conclusions

Prior work, funded by SRC, revealed that there are applications that tolerate faults in large portions of their state space, and continue to correct results. Our goal has been to enable users to share this information with their computing systems, and hence minimize unnecessary disruptions caused by soft errors, which are expected to be increasingly common in the future. We have made significant progress towards that goal. For example, we demonstrated this on the conjugate gradient algorithm, as directed by the government. Of course, as with any early research project, we also believe there is much more work to be done, before this technology can be integrated into the software widely used by Defense computational scientists.

Technology Transfer

We have distributed drafts of our assertion language to two DOD ACS program investigators, Prof. Vivek Sarkar at Rice University and Dr. David Bernholdt at Oak Ridge National Laboratory. Per the direction of John Daly, we also shared it with Erik deBenedictis of Sandia National Laboratory and Marti Bancroft, a contractor for another DOD organization. We demonstrated resilient execution of the HPCS Random Access benchmark at the DOD ACS workshop, July 17, 2014. We have also published in the scientific literature, as enumerated below. Finally, we have installed at NSA R3 an up-to-date release of the ROSE Compiler Infrastructure and an implementation of a selected set of the ROSE-based source-to-source transformations that support the resilience directives developed as part of the FAI-SAFE language at a computer cluster at the direction of the Government.

Future Plans

Our longer term goal is to broaden the space of faults we can handle without halting an application or the computing system its running on. We will would like to fully integrate and validate in a production environment the research artifacts developed here, namely, the integration of the ROSE compiler transformations with the user-provided codes in collaboration with the IFR system and release it as open-source for DOD and the broader community. We are searching for additional research support to continue these efforts.

Publications

- S. Hukerikar, P. Diniz, and R. Lucas, "Resilience for Exascale HPC Systems, A Programming Model Approach", SIAM Conf. on Parallel Processing for Scientific Computing, Feb. 2014, Portland, OR, USA

- S. Hukerikar, K. Teranishi, P. Diniz, R. Lucas, "Opportunistic Application-level Fault Detection through Adaptive Redundant Multithreading", Intl. Conf. on High Performance Computing & Simulation (HPCS 2014), July 2014, Bologna, Italy

- S. Hukerikar, K. Teranishi, P. Diniz, R. Lucas, "An Evaluation of Lazy Fault Detection based on Adaptive Redundant Multithreading", Eighteenth Annual IEEE High Performance Extreme Computing Conference 2014 (HPEC '14), Sept. 2014, Waltham, MA USA

- S. Hukerikar, P. Diniz, R. Lucas "Enabling Application Resilience through Programming Model based Fault Amelioration", in Proc. of the IEEE High Performance Embedded Computing Conference 2015, (HPEC'15), Sept. 2015, Waltham, MA USA

- S. Hukerikar, P. Diniz, R. Lucas, RoLex: Resilience Oriented Language Extensions for Exascale Computing, Journal of Supercomputing (2015) *(under review)*

- S. Hukerikar, P. Diniz, R. Lucas, Application level Fault Detection and Correction based on Adaptive Redundant Multithreading, Intl. Journal on Parallel Programming (2015) *(under review)*

- J. Lidman, S. McKee, D. Quinlan and C. Liao, An Automated Performance-Aware Approach to Reliability Transformations, Euro-Par 2014, August, Porto, Portugal.

Graduate Students Supported

Dr. Saurabh Hukerikar received his PhD in Electrical Engineering in the summer of 2015. His thesis was entitled "Introspective resilience for exascale high-performance computing systems". He is now a post doctoral scholar at the Oak Ridge National Laboratory.

Appendix 1
FAIL-SAFE Assertion Language V8.0

The FailSafe Assertion Language

Version 8.0

Hans P. Zima, Erik DeBenedictis, Jacqueline Chame, Pedro C. Diniz, Robert F. Lucas

May 18, 2015

Contents

Preliminary Note

This document contains the specification of Version 8 of the assertion language, representing a major revision of Version 7.0 distributed on June 20th, 2014. It includes new material on error control, introduces additional constructs and streamlines some of the previously introduced concepts, resulting in some changes of terminology.

The document continues to use an ad-hoc pseudo syntax whose main purpose is to serve as a framework for explaining the semantics of the assertion language. Any mapping of that syntax to a host language that preserves these semantics is considered valid.

1 Introduction

1.1 Fault Tolerance

Fault tolerance is one important aspect of a system's **dependability**, a property that has been defined by the IFIP 10.4 Working Group on Dependable Computing and Fault Tolerance as the "*trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers*".

A **threat** is any fact or event that negatively affects the dependability of a system. Threats can be classified as faults, errors, or failures; their relationship is illustrated by the *fault-error-failure* chain [1].

A **fault** is a defect in a system. Faults can be *dormant*—e.g., incorrect program code that is not executed—and have no effect. When *activated* during system operation, a fault leads to an **error**, which is an illegal system state. A fault inside a component is called *internal*; an *external* fault is caused by a failure propagated from another component, or from outside the system. Errors may be propagated through a system, generating other errors. For example, a faulty assignment to a variable may result in an error characterized by an illegal value for that variable; the use of the variable for the control of a for-loop can lead to ill-defined iterations and other errors, such as illegal accesses to data sets and buffer overflows. A **failure** occurs if an error reaches the service interface of a system, resulting in system behavior that is inconsistent with its specification.

In systems with an exascale computing capability we must assume that there exist faults and that errors will occur. The goal of the FailSafe system is to achieve **fault tolerance** by ensuring that despite the potential existence of errors the system will *never* enter a failure state.

1.2 Hard Correctness versus Soft Correctness

The traditional approach to program correctness has been based on the requirement that the program executes perfectly on a cycle-by-cycle basis, with numerical results mathematically precise except for possible rounding errors. We call this **hard correctness**. For algorithms such as sorting, which need to deliver a well-defined unique result, this represents the only valid approach to deal with correctness. However, for many algorithms this strict requirement can be relaxed by replacing it with **soft correctness**, a fidelity concept that defines the correctness of a result based on a user-perceived quality of the solution that tolerates some errors as long as they satisfy an associated validity criterion.

Early ideas in this direction originated from AI in the context of "soft computing" [12]; however the principal idea of qualitatively defining correctness at the algorithmic level applies to other applications as well, with examples including image processing, pattern-matching algorithms in bioinformatics, and iterative numerical algorithms yielding approximate results [7].

The adoption of soft correctness has immediate consequences for fault tolerance. Essentially, it leads to the notion of the *degree of correctness* of an algorithm, which is determined by the amount of tolerable error. For example, some errors may be completely ignored (such as a few erroneous pixels in a large image file), some errors occurring in a hierarchical control structure may be corrected at a higher level of abstraction, and even certain hardware component failures may be tolerated at the application level.

A major motivation for the distinction between hard and soft correctness is the significantly higher overhead in terms of execution time and energy consumption for enforcing hard correctness. As a consequence, it is often useful to partition a program into high-reliability and low-reliability sections. In high-reliability sections, program semantics demand a strictly enforced hard correctness approach. Reliability in such sections can be supported via software mechanisms such as correctness assertions, introspection, and redundant execution. In contrast, low-reliability sections can tolerate certain classes of errors, thus significantly reducing overhead. Errors occurring in such sections may be ignored or corrected at a higher level of abstraction, depending on algorithm-specific validity criteria. "Sandboxing" [3] is a method useful for certain numerical applications that supports such a distinction. It was originally used in computer security applications, where an untrusted "guest code" is executed in an isolated state space (the "sandbox"), which protects the rest of the application (the "host") from errors occurring during execution of the guest code. A recent case study illustrates this approach with FT-GMRES, a fault-tolerant Generalized Medium Residual algorithm for solving non-symmetric linear systems: the guest represents an unreliable inner solver for a linear system A.x=y, with a given termination criterion. The outer solver checks the solution vector for errors and determines if the result converges within the bounds specified by the algorithm; minor numerical perturbations are ignored. If the solution contains errors or diverges, it is disbanded and replaced with a valid value determined by information in the state space of the host.

1.3 Overview of the Assertion Language

The **assertion language (AL)** provides a set of constructs that can be embedded in a host language program for the purpose of checking the state of computations during runtime, expressing tolerance for certain kinds of program errors, and specifying relationships that can be used for dynamic program optimization.

In contrast to approaches such as the Java Assertion Language [10], AL is defined independently of a particular host language. However, given a specific host language, the details of AL semantics will to a certain extent rely on host language features such as expressions, naming, and scoping conventions.

AL constructs fall into four major categories: (1) assertions, (2) tolerance directives, (3) pragmas, and (4) error control features.

Assertions provide a means to express the knowledge that a propositional logic predicate over data should be satisfied at certain points during program execution. Predicates are classified as either status predicates or invariants. *Status predicates* define a precondition or postcondition for individual program statements, whereas *invariants* must be satisfied throughout extended regions.

A major purpose of assertions is their support for the hard correctness of an algorithms. For example, assertions used in the context of a sandbox may be used to determine if the potentially erroneous results of computation in a low-reliability region can be used at a higher level of abstraction. Assertions also provide some limited support for a *Design by Contract* capability [8, 9]. Specifically, status predicates allow reasoning about the partial correctness of program components via Hoare logic [4]: whenever a statement is executed in a state that satisfies its precondition then, if the execution of the statement terminates, the postcondition must hold immediately after its termination.

Tolerance directives instruct the system to ignore occurrences of certain classes of errors during program execution.

Pragmas can provide information about key relationships between energy consumption, performance, and reliability that can guide the system in optimizing an objective function at runtime, depending on the environment in which a program executes.

Finally, **error control** features provide tools for error recovery via the association of error-handling routines with AL constructs and source program components.

1.4 The FailSafe System

Exascale architectures will be characterized by heterogeneous processing and memory structures, millions of components, and deep memory hierarchies. Applications executing in such systems will need to balance the quest for performance optimization and minimization of energy consumption with a flexible approach to resilience that combines the enforcement of hard correctness for critical code components with an acceptable degree of fault tolerance for less critical code sections. The salient goal of the assertion language is to provide information that supports the **FailSafe** system, as outlined in Figure 1, in the implementation of strategies for the optimization of objective functions in this complex space.

A major component of the FailSafe system is the *Introspection Framework for Resilience (IFR)* [6]. The IFR encapsulates functionality related to the monitoring and analysis of errors and other special events that occur during program excution and reasons about related optimization and recovery strategies. At its heart is a powerful inference engine. The IFR generates information for the *Knowledge/Experience Database (KEX)*, which in an advanced version will include knowledge about the application domain, properties of the underlying hardware and software system, strategic optimization goals, and behavioral data extracted from actual executions. The information in the KEX evolves dynamically as an application is subject to iterative optimization steps during multiple compile/execute cycles; its input to the user, or to a future automated dynamic model evaluator, provides the opportunity to adapt the annotation of the source program for a new compile/execution iteration by taking into account the results of previous iterations and their analysis.

One of the implicit assumptions when discussing assertion languages is often that the management of its constructs is exclusively in the hands of the user. However, the above outline of IFR functionality in the context of KEX and powerful methods for compilation and runtime analysis, suggests that the compiler as well as the IFR and the runtime system may, in a future system, play an important role for the generation, management, and execution control of assertion language constructs, either autonomously or in cooperation with the user.

2 Regions

2.1 Syntax

1. *region* → *program-region* | *data-region*
2. *program-region* → *single-point-region* | *composite-region*
3. *single-point-region* → *statement-label* | *function-identifier*

4. *composite-region* → *function-region* | *assertion-region*
5. *assertion-region* → *statement-label* ": **region** " [*error-handler*] "{" *statement-sequence* "}"
6. *data-region* → *declaration-label* | *variable-identifier* | *memory-region-identifier*
7. *aggregate-region* → *aggregate-single-point-region* | *aggregate-data-region* | *aggregate-program-region*
8. *aggregate-single-point-region* → " **in** (" *single-point-region-list* ")"
9. *aggregate-data-region* → " **in** (" *data-region-list*")" [*region-constraint*]
10. *region-constraint* → *modifier* "(" *composite-region-list* ")" | *other-constraint*
11. *modifier* → " **excluding** " | " **restricted_to** "
12. *aggregate-program-region* → " **in** (" *composite-region-list*")" [*region-constraint*]

2.2 Regions and Their Association With Assertion Language Constructs

Each defining occurrence of an AL construct associates it with one or more regions, which we call the *target regions* of the construct. Regions exist in space and time: they are connected to certain components of the source program, such as variable or function declarations, and their period of existence is determined by the way in which these components are used during program execution. We use the term **lifetime** to refer to this period both for regions as well as for their associated AL constructs.

A **region** is either a program region or a data region. **Program region**s can be single point or composite regions. A *single point region* is either defined by a statement label or by a function identifier. In the first case, it identifies the statement determined by that label; in the second case, it represents *all* statements that invoke the function. If S denotes such a statement, then it is associated with two single points of execution: its *pre-execution point* denotes the point immediately before an execution of S, and its *post-execution point* the point immediately following an execution of S.

In contrast to single point regions, *composite region*s are defined by contiguous sets of statements. They can be function regions or assertion regions. The lifetime of a *function region* is determined by the period of time in which the program executes any invocation of that function. An *assertion region* is a labeled contiguous block of source code statements marked by the **region** keyword and enclosed between "{" and "}". Assertion regions must satisfy the single-entry condition. Their lifetime is any period of time in which program execution resides in that region.

A **data region** is either a *declaration region*, identified by the name of a source language declaration [1], or a *variable region*, identified by the name of a single variable. If $v_1, \ldots, v_n$ are the variables declared in a declaration region, then it represents the set of all variable regions for the $v_i, 1 \leq i \leq n$.

During program execution, a variable region will result in the creation of a set of data objects. Depending on the declaration of the variable such objects may be of a simple type, pointers, or aggregates such as arrays, structures or class instances. The lifetime of a variable region is the period of time during which any associated objects exist during program execution. The lifetime of a declaration region is the union of the lifetimes of the associated variable regions. For tolerance directives (see Section 5), specially managed error-tolerant memory regions are also considered as data regions.

[1] Here we assume that the language provides syntax for labeling a declaration.

The association of an AL construct with its target regions can be established in two ways: first, *by context*, i.e., by placing the construct in the source program text adjacent to a program object, which then becomes its single target [2]. Secondly, as a syntactic convenience, the defining occurrence of an AL construct can be specified separately from its target regions in the source program by providing an explicit link to its associated *aggregate-region*. In general, such a link is of the form:

$$\textbf{in}\,(R_1, ...R_n)\;[C]$$

where $n \geq 1$, $R_1, ...R_n$ denote regions, and C is an optional *region constraint*. All regions $R_1, ...R_n$ must be of the same type, i.e., single-point regions, composite regions, or data regions. A region constraint can be used to exclude some implicit components, such as nested functions or assertion regions, from the aggregate region. For example:

float A, B, C;
...
atype function f(...) {...};
...
assert (p(**allvars**)) **in** (A,C) **excluding** (f);

Here, the invariant predicate p is asserted to hold for the assertion's target variables A and C throughout their lifetimes, except when program execution resides in an invocation of function f (**allvars** refers to every variable with which the assertion is associated).

The detailed rules governing the association of AL constructs with their targets will be discussed in the relevant sections.

3 Error Control

3.1 Syntax

1. *error-control* → *error-handler* | *redundancy-management-directive*
2. *error-handler* → *basic-error-recovery* | *standard-error-recovery* | *composite-region-error-recovery*
3. *basic-error-recovery* → " **error** (" *basic-error-handler* ")"
4. *basic-error-handler* → *function-invocation*
5. *standard-error-recovery* → *basic-error-recovery* | " **error** ("*class-specific-recovery*$^+$ [*else-clause*] ")"
6. *class-specific-recovery* → *error-class* "→" *basic-error-handler*";"
7. *error-class* → **numerical_error** | **NaN** | **Inf** | *floating-point-representation-error* | *integer-representation-error* | **SECDED** | **crash** | **predicate_violation** | **tolerance_violation** | *other-predefined-error-class* | *user-defined-error-class* | **ANY**
8. *else-clause* → " **else** " *basic-error-handler*

[2] The detailed rules for contextual association are implementation dependent.

9. *composite-region-error-recovery* → *standard-error-recovery repetition-directive* [*save-directive*]
10. *repetition-directive* → " **repeat** ("*rep-count* ["," *restore-directive*] [*"," final-error-recovery*] ")"
11. *rep-count* → *integer-expression*
12. *restore-directive* → " **restore** (" *data-region-list* ")"
13. *final-error-recovery* → " **finally** (" *basic-error-handler* ")"
14. *save-directive* → " **save** (" *data-region-list* ")"
15. *redundancy-management-directive* → *redundancy-degree decision-criterion* [*standard-error-recovery*]
16. *redundancy-degree* → " **double** " | " **triple** "
17. *decision-criterion* → "(" *variable-list* ")"

3.2 Overview

Each error occurring during the execution of a program belongs to a specific *error class*. For system-defined error classes, such as **numerical_error** , default error handlers are provided. Future versions of AL are expected to provide a capability for user-defined error classes and their management.

Error control in AL includes two sets of features: the first set, subsumed under the term *error handler*, allows the specification of recovery mechanisms for errors occurring in the associated AL or source program construct. Three levels of increasing complexity are distinguished: *basic error handlers* deal with an error by calling a function that executes a recovery routine. This is the only available option for status predicates. *Standard error recovery* includes basic error handlers plus a capability for selecting an error response depending on the error class; it can be used in the context of data regions and composite regions. Finally, *composite region error recovery* supports additional features specially oriented towards composite regions including repeated execution of the region and control for the saving and restoring of data structures.

The second set of error control features of AL is focused on preventing and/or correcting errors occurring in composite regions. It provides *redundancy management directives* that execute a composite region two or three times and evaluate the correctness of results based upon a decision criterion.

3.3 Error Classes

Below we characterize briefly some of the system-defined error classes.

1. **numerical_error** : this is the class of *all* kinds of numerical errors, including the four classes descibed below.
2. **NaN** : In the IEEE-754 standard, **NaN** (Not a Number) represents a non-numerical value occurring in a context that requires a number.
3. **Inf** : In the IEEE-754 standard, **Inf** (Infinity) represents the occurrence of an overflow in a numerical computation.
4. **Floating-Point Representation Error**

5. **Integer Representation Error**

6. **SECDED** : Single-error correction, double-error detection.

7. **crash** : Processor crash.

8. **predicate_violation** : Evaluation of a status predicate or invariant during its lifetime yields **false** (see Section 4).

9. **tolerance_violation** : Evaluation of a tolerance clause during its lifetime yields **false** (see Section 5).

10. **ANY** : An error belonging to any class.

3.4 Error Handler

3.4.1 Basic Error Recovery

Basic error recovery results in the invocation of a *basic error handler*, which is a system- or user-defined function. This function call may contain arguments that provide a more detailed characterization of the error as well as additional information about program components that may have contributed to the error, such as key variables involved in the erroneous computation.

Example

x=f(y) **assert** (**pre** ($y < U$)) **error** (range_violation(y,U));

In this example, the predicate $y < U$ is a precondition associated with the assignment statement $x = f(y)$, which needs to be satisfied immediately before an execution of the assignment. Its violation results in the invocation of the basic error handler *range_violation*.

3.4.2 Standard Error Recovery

Standard error recovery can be used to address errors that occur during the execution of a data or composite region. In addition to basic error recovery it provides a capability for specifying recovery depending on the error class:

error ($c_1 \rightarrow h_1$;
...
$c_n \rightarrow h_n$;
[**else** h_{n+1}])

where,

- $n \geq 1$ is the number of error classes for which recovery is specified explicitly
- $c_i, 1 \leq i \leq n$, denotes an error class
- $h_i, 1 \leq i \leq n$, denotes a basic error handler associated with error class c_i

- h_{n+1} denotes the optional basic error handler for any error with a class different from all of the $c_i, 1 \leq i \leq n$

Assume R to be the region with which this construct is associated, and that an error of class c has been detected during execution in R. Then c is sequentially compared with $c_1, c_2, ...$ If $c_i, 1 \leq i \leq n$, matches c, then h_i will be activated. If c is different from all $c_i, 1 \leq i \leq n$, and an else clause is specified, then handler h_{n+1} will be activated. In the absence of an else clause a system-specific response will be initiated in this case [3].

3.4.3 Composite Region Error Recovery

A composite error recovery specification is of the form:

error ($c_1 \ \rightarrow \ h_1$;
...
$c_n \ \rightarrow \ h_n$;
[**else** h_{n+1}])
repeat (maxrep [, **restore** ($v_{j_1}, ...v_{j_p}$)] [, **finally** (h)])
[**save** ($v_1, ...v_m$)]

where,

- n, c_i, and h_i have the same meaning as above
- *maxrep* is an integer expression with a value ≥ 1 that specifies the maximum of repetitions of R to be performed in the case of error.
- the $v_{j_1}, ...v_{j_p}$ in the optional restore directive denote a subset of the variables $v_1, ...v_m$ specified in the associated save directive
- h denotes a basic error handler
- $v_j, 1 \leq j \leq m$, with $m \geq 1$, in the optional save directive specifies a program variable

This construct is executed as follows:

1. R is initially activated, after executing the save directive if present.
2. If the execution of R is error-free, normal program execution continues with the statement following R.
3. If during the execution of R an error occurs, then a response as described above in Section 3.4.2 is initiated. If this response is able to successfully process the error in the sense that an effect equivalent to a correct execution of R is achieved, normal program execution is continued with the statement following R.

   Otherwise, a repeated execution of R is initiated, after restoring the values of variables specified in the restore directive, if present. If a restore directive is not specified, but a save directive has been executed at the beginning, all variables specified in the save directive will be restored at that point.

[3] The semantics of this construct is slightly different if used in the context of tolerance directives, see Section 5.

4. Step 3 is repeated until R has been executed at most *maxrep* times. If, at the *maxrep*-th execution, no error occurs, normal program execution continues. Otherwise, if present, the handler, h, specified in the *final error recovery* is activated. If no final error recovery is specified, a system dependent error handler will be invoked at this point.

If a save directive is part of the construct, then the variables $v_j, 1 \leq j \leq m$, will be saved by the system immediately before the execution of R is initiated [4].

Example

```
R1: region R1 error ( SECDED  → secded_handler(...);
                      numerical_error  → numeric_handler(...);
                      else  other_handler(...))
                repeat (2, finally (final_handler(...)));
{S1;S2;...}
```

If an error occurs during the initial execution of region $R1$, the following actions are performed:

- If the error is of class **SECDED** or **numerical_error** , then *secded_handler* or *numeric_handler* will be respectively activated. If an error of a different class occurs, then *other_handler* will be activated. If the error handler can repair the problem, then normal execution continues with the statement following $R1$. Otherwise, the execution of $R1$ is repeated.
- After the first repeated execution of $R1$, the same actions as in Step 1 above are taken.
- If a second repeated execution of $R1$ is necessary, and no error occurs during that execution, normal program execution continues. If an error occurs during that execution, the basic error handler *final-handler(...)* is activated.

3.5 Redundancy Management Directives

Error control features in AL include a preliminary version of **redundancy management directives (RMD)**s, which control the redundant execution of a composite region, R, subject to two constraints [5]: (1) R satisfies the single-entry single-exit condition, and (2) the execution of R does not result in any side effects for non-local variables. A region satisfying these constraints is called an *RMD region*; it can be associated with an optional standard error recovery construct.

A key component of an RMD is the *decision criterion*: it specifies a nonempty list of *decision variables* whose values at the end of redundant executions determine the decision on the correctness of the region's execution.

Two kinds of RMDs are distinguished: double redundancy directives, and triple redundancy directives. Let R denote a composite region to which one of the directives is applied. Then the effect of RMDs can be described as follows:

[4] The execution of the save directive may result in considerable overhead in terms of performance, memory use, and energy consumption, in particular if large data structures need to be copied. This problem may be alleviated by providing more flexibility for the choice of data to be saved and/or restored via additional control features.

[5] Loosening these restrictions may be possible in certain contexts. This applies specifically to the single-exit constraint, which is not required for extended basic blocks.

1. **Double Redundancy Directive:** R will be executed twice, applied to the same input and without any intermediate computations affecting the results of these executions.

   If at least one of these executions encounters an error before terminating, then the execution of R is considered erroneous, and a recovery action is taken upon the first detection of such an error.

   If both executions of R terminate without error, and the values of all decision variables after termination are identical for both executions, then the execution of R is considered error-free. Otherwise, the execution of R is considered erroneous.

2. **Triple Redundancy Directive:** R will be executed thrice, applied to the same input and without any intermediate computations affecting the results of these executions.

   If at least two executions of R encounter an error before terminating, then the execution of R is considered erroneous, and a recovery action is taken upon the first detection of such an error.

   If at least two executions terminate without error, and the values of all decision variables after termination are identical for both executions, then the execution of R is considered error-free. Otherwise, the execution of R is considered erroneous.

4 Assertions

4.1 Syntax

1. *assertion* → *assert-clause* [*assertion-target*] [*error-handler*]
2. *assert-clause* → "**assert** (" *predicate-list* ")"
3. *predicate* → *status-predicate* | *invariant*
4. *status-predicate* → *precondition* | *postcondition*
5. *precondition* → "**pre** (" *boolean-expression* ")"
6. *postcondition* → "**post** (" *boolean-expression* ")"
7. *invariant* → *boolean-expression*
8. *assertion-target* → *aggregate-region*

4.2 Overview

The specification of an **assertion** consists of up to three components: (1) a mandatory assert clause, (2) an optional assertion target, and (3) an optional error handler.

An assert clause contains between one and three predicates, including at most one precondition, one postcondition, and one invariant. If an error handler is provided for an assertion, it applies to all its predicates.

4.3 Assert Clauses

An **assert clause** consists of the keyword **assert** followed by a parenthesized list of **predicates**, which are propositional logic expressions over data. Depending on how a predicate is used, we distinguish between *status predicates* and *invariants*. A status predicate is associated with single point regions; it can be either a *precondition* or a *postcondition* and must be satisfied respectively at the pre-execution or the post-execution points of these regions (see Section 2). An *invariant* can be associated with either data regions—and called a *data invariant* in this case—or with composite regions—and called a *program invariant*.

The core component of a predicate is a boolean expression (which in some cases may be tweaked due to special context requirements). We call this expression an *assertion expression*. It may contain pure functions and must be side-effect free.

Let E denote the an assertion expression, and let a *predicate application location (PAL)* be any location in the program, where E can be legally applied during execution within the assertion's lifetime. For all identifiers referenced in the assertion—including identifiers in E, region identifiers, and any identifiers referenced in an error handler—the links to their defining occurrences are established at the point of the assertion's defining occurrence. If loc is an arbitrary PAL, then all these links must be defined at loc, and the value of E is computed using the values of its identifiers at this location. If that evaluation yields **true**, the predicate is said to be *satisfied* at loc, otherwise it is said to be *violated* at loc, resulting in an error of class **predicate_violation**. If any of the variable links in E is undefined at loc, or if a variable reference yields a corrupted value, then the application of the predicate in this location is *undefined* and the expression cannot be properly evaluated, resulting in an error. This class of error is a special case of a **predicate_violation**.

4.4 Rules for the Association Between Predicates, Regions, and Error Handlers

The association between predicates, associated regions, and error handlers is subject to the following rules:

1. **Status Predicates:** An assertion specifying a status predicate can only be associated with an aggregate single-point region and a basic error handler.

   For any single-point region, at most one precondition and at most one postcondition can be specified.

2. **Data Invariants:** An assertion specifying a data invariant can only be associated with an aggregate data region. It can be combined with standard error recovery.

   Any data region can be associated with at most one data invariant.

3. **Program Invariants:** An assertion specifying a program invariant can only be associated with an aggregate program region. It can be combined with a composite region error recovery.

   Any composite region can be associated with at most one precondition, at most one postcondition, and at most one program invariant.

Multiple predicates for one assertion target can be specified in one or more assertions. Splitting a multiple-predicate assertion into separate assertions allows the violation of different predicates to be recovered by different error handlers.

4.5 Built-In Predicates for Error Detection

4.5.1 Check_Predicate

Consider an invariant, **assert** (E), associated with a variable, v. If nothing else is said, the system—i.e., the compiler and runtime system—is responsible for validating the invariant throughout the lifetime of the assertion. Depending on the context in which the assertion is used, this may result in a significant runtime overhead. AL provides a way to explicitly specify at which points during program execution the assertion expression needs to be evaluated via a built-in function **check_predicate** (v), whose invocation results in the evaluation of E. This function can be used in a status predicate as shown in the example below:

atype v **assert** (E) **no_check** ;
...
S: v=g(x,y,z) **assert** (**post** (**check_predicate** (v))) **error** (h(...);

The above declaration defines an invariant assertion, **assert** *(E)*, that is contextually associated with the declaration of variable v. The keyword **no_check** attached to this assertion is meant to suppress its automatic validation by the system.

The status predicate associated with statement S uses the built-in function **check_predicate** (v) as a postcondition. This function is invoked immediately after the completion of the assignment $v = g(x, y, z)$; it results in the evaluation of the boolean expression E at that point.

Note that since AL requires at most one invariant to be associated with a data region, the reference to v in the status predicate determines this invariant unambiguously.

4.5.2 Check_Tolerance

A similar function is provided for the explicit validation of a tolerance expression associated with a variable, v (see Section /reftolerance). It is invoked in the form

check_tolerance (v)

and yields **true** iff the tolerance expression for v is satisfied.

4.5.3 Check_Error

An invocation of the built-in predicate **check_error** (c), where c denotes an error class, results in a value of **true** iff at the time of execution an error of class c has been detected by the system.

4.6 Examples

Status Predicate: Example 1

x=f0(y) **assert** (**post** $(x < U)$) **error** (h1(x,y,U));
L1: z=g(x);
...

This example shows an assertion associated by context with the single-point region defined by the assignment statement $x = f0(y)$. The assertion specifies a postcondition, i.e., the assertion expression $x < U$ needs to be evaluated immediately after the execution of the assignment $x = f0(y)$, and before the statement $z = g(x)$ at label $L1$ is executed. In the case that the assertion fails, the basic error handler *h1(x,y,U)* is activated.
If $L1$ is the target of a control transfer, the above assertion may or may not be satisfied when control is transferred to $L1$ from an outside location. In order to assure that the assertion is always satisfied before execution of the statement labeled by $L1$, in whatever way this statement is reached, the assertion would have to be formulated as a precondition for that statement, as shown below:

x=f0(y);
L1: z=g(x) **assert** (**pre** ($x < U$)) **error** (h1(x,y,U));
...

The above program could also be formulated by using an explicit association:

x=f0(y);
L1: z=g(x);
...
assert (**pre** ($x < U$)) **error** (h1(x,y,U)) **in** (L1);

Status Predicate: Example 2

float function f1(**float** x,y,z);
{... };
...
assert (**pre** (p(a,b,c)), **post** (q(v,...))) **in** (L2);
...
L2: v=f1(a,b,c);

In this example, two status predicates – a precondition as well as a postcondition – are explicitly associated with the single point region identified by the label *L2*.

Now consider a slightly more general example by explicitly associating a postcondition, u, with the function, $f1$. This postcondition applies to *all* invocations of $f1$, and therefore specifically to its invocation within the assignment at label *L2*:

float function f1(**float** x,y,z); **assert** (**post** (u(...)));
{...};
...
assert (**pre** (p(a,b,c)), **post** (q(v,...))) **in** (L2);
...
L2: v=f1(a,b,c);

Now the postcondition $u(...)$ will be evaluated immediately after completion of the invocation $f1(a, b, c)$. Following that, after execution of the assignment to v, the postcondition for the call, $q(v, ...)$, will be evaluated.

Status Predicate: Example 3

assert (**post** (A(i)$\leq$ B(i))) **error** (h3(i,A(i),B(i))) **in** (LL);
...
LL: CALL MPI_RECV(A(i),1,MPI_REAL,myrank+1,...);

Here a status predicate is associated explicitly with the statement identified by the label *LL*, which is an MPI library call that receives the value of $A(i)$. The expression $A(i) \leq B(i)$ is a postcondition that must be satisfied immediately after the execution of that statement. For the case that it fails the error handler *h3(...)* will be activated.

Status Predicate: Example 4

BB1: **region**
{ /* basic block statement sequence */ };
...
assert (**post** ($(v1 \leq upb) \wedge (v2 \geq lwb) \wedge (v2 \leq v1)$)) **in** (BB1);

Here, the assertion expression $(v1 \leq upb) \wedge (v2 \geq lwb) \wedge (v2 \leq v1)$ serves as a postcondition for the execution of the assertion region labeled by BB1.

Status Predicate: Example 5

LOOP1: **while** cexpr
{
/* basic block statement sequence */
assert (**post** ($x \leq y$)) **error** (...);
}

In this example, the postcondition $x \leq y$ is textually associated with the last statement of the while loop's body. It will be evaluated every time the body of the while loop is executed, immediately after the execution of the last statement in the body. As a consequence, this expression represents an invariant that must be satisfied *after* each iteration of the loop, and *after* its termination. Note, however that this predicate need not be satisfied *within* the loop body.

Status Predicate: Example 6

L1: x=g(y);
...
assert (**pre** (p1(x))) **error** (...) **in** (L1);
...
assert (**pre** (p2(y))) **error** (...); **in** (L1) /* ILLEGAL */

This is illegal, since no two preconditions can be associated with a single program region (see Section 4.4).

Invariant: Example 1

```
 R: region  assert (x ≤ y)
{ while cexpr
  {
  /* basic block statement sequence */
  }
}
```

The difference between this example and Status Predicate Example 5 is that here the while loop has been enclosed in an assertion region, R, and that the post condition for the last statement has been replaced by an invariant that needs to be satisfied throughout the region.

Invariant: Example 2

```
class cyclic_buffer
{ int n;
  double B(n);
  int c_cycles, p_cycles;
  ...
}
assert ((c_cycles ≤ p_cycles) ∧ (p_cycles ≤ c_cycles + n)) error (sync_error_handler(...));
```

The assertion in this example specifies an invariant contextually associated with the class *cyclic_buffer*. Its lifetime is the union of lifetimes of objects that are instantiations of *cyclic_buffer*. The invariant expresses a standard constraint for producer and consumer processes operating asynchronously on a cyclic buffer, $B(n)$, where n is the size of the buffer and c_cycles and p_cycles respectively represent the number of cycles the consumer and producer have already executed.

Assume now that statement S below has updated c_cycles, as a consequence of consuming an item from the buffer. Then the postcondition associated with S performs an explicit check of the invariant for $cyclic\_buffer$ defined above:

```
S: new_item = consume_an_item(...) assert ( post ( check_predicate (cyclic_buffer))) error (h(...));
```

The basic error handler *h(...)* is activated if the post condition for statement S is violated. This overrides the error handler, *sync_error_handler(...)*, provided for the invariant in the context of the class declaration.

Invariant: Example 3

```
int ncycles; assert (0 ≤ ncycles ≤ maxcycles);
```

Here we use a contextual association for the specification of an invariant for the variable *ncycles* that expresses bounds for its values that must be respected throughout the lifetime of the variable. This could be rewritten using an explicit association:

int ncycles;
...
assert $(0 \leq ncycles \leq maxcycles)$ **in** $(ncyles)$;

Below we modify this example by limiting the lifetime of the invariant to the time program execution resides in the composite regions $R1$ and $R2$:

int ncycles;
...
assert $(0 \leq ncycles \leq maxcycles)$ **in** $(ncycles)$ **restricted_to** (R1,R2);

Invariant: Example 4

int ncycles, mcycles **assert** $(0 \leq \mathbf{allvars} \leq maxcycles)$;

This is similar to the previous example, except for the fact that the invariant is now valid for both of the variables mentioned in the declaration, i.e., *ncycles* as well as *mcycles*, due to the use of the quantifier **allvars** .

Invariant: Example 5

int i, j, k;
...
int * ip; **assert** $((range(ip) = (\&i, \&k)) \wedge (lwb \leq *ip \leq upb))$

The integer pointer variable ip is restricted to point to the variables i and k; in addition, the values of the variables pointed to by ip must always lie between the bounds lwb and upb.

Invariant: Example 6

float function f1(**float** x,y,z);
error (**numerical** $\rightarrow$ hw1(...);
SECDED $\rightarrow$ hw2(...);
else hw3(...))
repeat (k);
assert (w(...));
assert (**post** (u(...))) **error** (hu(...));
{...};

...
assert (**pre** (p(a,b,c)), **post** (q(v,...))) **in** (L2);
...
L2: v=f1(a,b,c);

This is an extension of the second version of Status Predicate Example 2. Everything that has been said there is still valid. In addition, an error handler and an invariant, $w(...)$, have been associated with the function. This means that throughout any invocation of the function, $w(...)$ needs to be satisfied. If, during an invocation, an error of class **numerical** or **SECDED** occurs, the error handlers *hw1(...)* or *hw2(...)* are respectively activated. If an error of any other class occurs, error handler *hw3(...)* is activated. If such an recovery action is unsuccessful, the function activation is repeated up to $k \geq 1$ times (see Section 3).

5 Tolerance Directives

5.1 Syntax

1. *tolerance-directive* → *tolerance-clause* [*tolerance-target*] [*standard-error-recovery*]
2. *tolerance-clause* → *variable-tolerance-clause* | *global-tolerance-constraint*
3. *variable-tolerance-clause* → *error-tolerance-clause* | *tolerant-memory-clause*
4. *error-tolerance-clause* → " **tolerate** (" *tolerance-expression* ")"
5. *tolerance-expression* → *boolean-violation-permit-expression*
6. *violation-permit* → "(" *threshold* "," *error-class* ")"
7. *threshold* → *integer-expression* | " **accumulated** (" *integer-expression* ")" | " **access_percentage** (" *number-of-variable-accesses*, *error-percentage* ")" | **all**
8. *number-of-variable-accesses* → *integer-expression*
9. *error-percentage* → *floating-point-expression*
10. *tolerant-memory-clause* → " **tolerant memory** (" *integer-expression* ")"
11. *tolerance-target* → *variable-tolerance-target* | *global-constraint-tolerance-target*
12. *variable-tolerance-target* → *aggregate-data-region*
13. *global-constraint-tolerance-target* → *aggregate-data-region* | *aggregate-program-region*
14. *global-tolerance-constraint* → " **constrain_tolerance** (" *boolean-expression* ")"

5.2 Overview

A **tolerance directive** instructs the system to ignore certain errors that occur during the manipulation of data. Its specification contains up to three components: (1) a mandatory tolerance clause, (2) an optional tolerance target, and (3) an optional standard error recovery.

A tolerance clause is either a variable tolerance clause or a global tolerance constraint. The target of a *variable tolerance clause* is an aggregate data region, and the tolerance clause in this case expresses the error permissions for the data objects belonging to that region. A *global tolerance constraint* can be associated with an aggregate data region or an aggregate program region; it specifies a constraint over a set of variable tolerance clauses in that region.

If, during execution in its target region, the condition expressed in a tolerance clause is violated, an error of type **tolerance_violation** is raised. An explicit check for the validity of a tolerance clause can be expressed via the built_in function **check_tolerance** (v) (see Section 4.5).

5.3 Variable Tolerance Clauses

We distinguish two different types of variable tolerance clauses: first, an *error tolerance clause* lists specific error classes that should be tolerated, together with a threshold limiting the number of permitted violations. Secondly, a *tolerant memory clause* directs the compiler to allocate "tolerant memory" for the associated data objects. It results in the suppression of a limited number of memory errors.

5.3.1 Error Tolerance Clause

An **error tolerance clause** consists of the keyword **tolerate** , followed by a parenthesized *tolerance expression*, which is a boolean expression formulating a logical condition over violation permits. Each *violation permit* consists of an error class together with a threshold; it determines a boolean value as explained below.

Assume that the error tolerance clause in consideration is associated with variables v_1,...v_n, where $n \geq 1$, and (t, c) is a violation permit, where t is a threshold, and c an error class. Depending on t, the meaning of the violation permit (t, c) is defined as follows:

1. t is an integer expression
   In this case, the value of t must be ≥ 0 and the violation permit is satisfied iff for *each* of the variables $v_i, 1 \leq i \leq n$, there are at most t occurrences of error class c. If $t = 0$, then the effect of the violation permit is the same as if it was absent, i.e., any occurrence of an error of class c in any of the variables results in an error.

2. $t =$ **accumulated** (t')
   Here, t' is an integer expression with a value ≥ 0. The violation permit is satisfied iff the number of occurrences of error class c, accumulated over *all* of the variables $v_i, 1 \leq i \leq n$, is at most t'.

3. $t =$ **access_percentage** (k, r)
   Here, k is an integer expression with a (usually large) positive value, and r is an expression yielding a floating point number in the range $0 \leq r \leq 100$. The violation permit is satisfied iff for *each* of the variables $v_i, 1 \leq i \leq n$, and for any k subsequent accesses to that variable at most $rk/100$ errors of class c occur.

4. $t = \mathbf{all}$
   In this case, there is no limit to occurrences of error class c in any of the $v_i, 1 \leq i \leq n$: the violation permit is always satisfied, independent of how many errors of that type occur.

5.3.2 Examples for the Error Tolerance Clause

Error Tolerance Clause: Example 1

float A, B, C; **tolerate** ((n1, **Inf**) $\wedge$ (n2, **NaN**));
float D; **tolerate** ((n3, **SECDED**));
float E, F; **tolerate** ((**accumulated** (n4), **ANY**));
float *ip1; **tolerate** ((**access_percentage** (1000,2), **SECDED**));
float G; **tolerate** ((**all** , **ANY**));

Here, let *n1*, *n2*, *n3*, and *n4* denote integer expressions, all yielding values ≥ 0. The error tolerance clauses specified above have the following meaning:

- During the manipulation of the floating point variables *A*, *B*, and *C* *n1* **Inf** and *n2* **NaN** errors are tolerated for each of these variables. No other errors are tolerated.
- The manipulation of the floating point variable *D* permits *n3* **SECDED** errors. No other errors are tolerated.
- The manipulation of variables *E* and *F* tolerates an accumulation of at most *n4* arbitrary errors in both variables together, i.e., the number of errors occurring during the manipulation of *D* plus the number of errors occurring during the manipulation of *F* must not exceed *n4*.
- The manipulation of the floating point values of variables pointed to by *ip1* tolerates at most 2% **SECDED** errors for any 1000 successive accesses to these variables.
- The tolerance for variable *G* is the most encompassing: all errors of any type that occur during manipulation of *G* are tolerated.

Error Tolerance Clause: Example 2

float A, B, C;
float D;
...
tolerate ((n1, **Inf**) $\wedge$ (n2, **NaN**)) **in** (A, B, C);
tolerate ((n3, **SECDED**)) **in** (D);

The semantics of this example are the same as for the first two lines in the previous one. The difference is in the syntax, which allows for the separation of declarations and the associated error tolerance clauses.

Error Tolerance Clause: Example 3

float H;
...
tolerate ((**all** , **Inf**)) **in** (H) **restricted_to** (f1);
...
float function f1(...) { /* function body */ };

Here, the tolerance directive associated with variable *H* is imposed during program execution in each invocation of function *f1*, and is not in effect elsewhere. All occurrences of **Inf** errors are tolerated.

Error Tolerance Clause: Example 4 – Numerical Representation Errors

float J **tolerate** ((n1, **mantissa** [7:0]));
int K **tolerate** ((n2, **bits** [31:16]));

Let $n1$ and $n2$ denote integer expressions with values ≥ 0. The manipulation of the floating point variable *J* tolerates *n1* errors in the least significant 8 bits of the mantissa. No other errors are permitted.

The manipulation of the integer variable *K* tolerates *n2* errors in the upper 2 bytes of its representation. No other errors are permitted.

Error Tolerance Clause: Example 5 - Standard Error Recovery in a Data Region

float A **tolerate** ((n1, **NaN**) $\wedge$ ((n2, **Inf**) $\wedge$ (n3, **SECDED**))
error (**NaN** $\rightarrow$ NaN_violation_handler(n1,...);
Inf $\rightarrow$ Inf_violation_handler(n2,...);
SECDED $\rightarrow$ secded_error_handler(n3,...);
else other_handler(error-class,...));

Here, let *n1, n2*, and *n3* denote integer expressions, all yielding values ≥ 0. The tolerance directive specified above has the following meaning: During the manipulation of the floating point variable *A*, *n1* **NaN** arithmetic errors, *n2* **Inf** arithmetic errors, and *n3* **SECDED** errors are tolerated. No other errors are tolerated.

Two types of errors can occur in this constellation: first, there may be errors that are *not* listed in the violation permit, i.e., errors different from **NaN** , **Inf** , and **SECDED** . These are called *non-tolerated errors*. Secondly, there may occur more **NaN** , **Inf** , and/or **SECDED** errors than are tolerated by the directive. Error handling is able to distinguish these cases and treat them separately as follows:

- The occurrence of more than $n1$ **NaN** errors is handled by the *NaN_violation_handler*.
- The occurrence of more than $n2$ **Inf** errors is handled by the *Inf_violation_handler*.
- The occurrence of more than $n3$ **SECDED** errors is handled by the *secded_error_handler*.
- The occurrence of any other error results in the activation of error handler *other_handler(error-class,...)*.

Note that the semantics of the error clause is slightly different from that discussed in Section 3 in that a tolerated error is treated in the same way as no error.

Error Tolerance Clause Example 6 - Multiple Error Handlers for Multiple Variables

float D,E; **tolerate** ((**accumulated** (n4), **numerical_error**))
error (**numerical_error** → numerical_error_handler(**accumulated** ,n4,D,nD,E,nE,...);
else other_handler(error-class,...));

Let *n4* denote an integer expression yielding a value ≥ 0. The tolerance directive has the following meaning:

The manipulation of the variables *D* and *E* tolerates an accumulation of at most *n4* numerical **NaN** errors in both variables together, i.e., the number of such errors occurring during the manipulation of *D* plus the number of errors occurring during the manipulation of *E* must not exceed *n4*. In the case this limit is violated, the numerical error handler is activated. Its arguments include the limit, *n4*, each affected variable—D, E—together with the number of error occurrences caused by that variable—*nD* and *nE* respectively. As before, the occurrence of any other, i.e., non-tolerated, error triggers the activation of *nte_handler*.

5.3.3 Tolerant Memory Clause

A **tolerant memory clause** consists of the keyword **tolerant memory** followed by an integer expression, which must yield a positive value, t.

This directive specifies that memory for the associated data objects is to be allocated in an *error-tolerant memory region*. Such a region is subject to a special set of rules: the operating system keeps track of uncorrectable errors during execution in this region, but takes no further action as long as the number of errors does not exceed t.

The allocation of data objects in an error-tolerant memory region can be performed in the C programming language by calling a special version of the *malloc* routine.

5.3.4 Example for Tolerant Memory Clause

atype A; **tolerant memory** (t)
...
A = tolerant_malloc(sizeof(**atype**));

Variable *A* is declared of type **atype**. The associated directive specifies *A* to be allocated in an error-tolerant memory region, with a threshold of maximum *t* errors to be tolerated. The call to the special routine *tolerant_malloc* performs the requested allocation.

5.4 Global Tolerance Directives

The features provided by variable tolerance directives do not offer an easy way to specify a logical condition involving *multiple* such directives. This can be achieved via a **global tolerance constraint**. We illustrate the use of such a constraint with an example:

HPC_Sparse_Matrix * A;
float * x, *y;
...

tolerate ((1,SECDED)) **in** (*A, *x, *y);
...
constrain_tolerance (*A **xor** *x **xor** *y)

In this example, an identical variable tolerance clause is specified for the values pointed to by A, x, and y. The global tolerance constraint imposes an additional restriction in that it allows a SECDED error only to occur for at most one of these variables (**xor** represents the exclusive or operator).

6 Pragmas

6.1 Syntax

1. *pragma* → *pragma-clause* [*pragma-target*] [*standard-error-recovery*]
2. *pragma-clause* → *voltage-clause* | *other-pragma-clause*
3. *voltage-clause* → " **voltage_options** ("*voltage-variable* "," *probability-specification-list* ")"
4. *probability-specification* → *undetected-error-probability-specification* | *detected-error-probability-specification* | *any-error-probability-specification*
5. *undetected-error-probability-specification* → " **p_undetected_errors =**"*expression*;
6. *detected-error-probability-specification* → " **p_detected_errors =**"*expression*;
7. *any-error-probability-specification* → " **p_errors =**"*expression*;
8. *pragma-target* → *aggregate-data-region* | *aggregate-program-region*

6.2 Overview

The main purpose for introducing pragmas in this document is as a placeholder for future extensions of the language. In this version of AL, our only focus is on a special class of pragmas that support control of the tradeoff between energy and reliability.

The connection between energy efficiency and reliability can be exploited at a hierarchy of architecture and software levels. In essentially every computer in production today, a logic circuit can be designed for the control of power consumption and measured for error rate. Managing this tradeoff should be possible when new devices intended to extend Moore's Law are perfected. The power supply to these devices could be generated by a Digital to Analog Converter (DAC). By changing the power supply voltage, the speed, power, and error rate of devices could be changed in nanoseconds. It would be possible to build a computer where such changes could be applied to composite program regions under program control thus leading to algorithms that would be able to control the tradeoff between reliability, energy efficiency, and performance, and to more closely approach energy efficiency limits.

Pragmas are specified using a structure similar to that used for assertions and tolerance directives, as discussed in Sections 4 and 5. Their specification consists of up to three components: (1) a mandatory pragma clause, (2) an optional pragma target, and (3) an optional standard error recovery. In general, a *pragma target* can be either an aggregate data region or an aggregate program region. In both cases, a standard error recovery can be combined with the construct.

6.3 Voltage Pragmas

Voltage pragmas are associated with aggregate program regions. They provide information about the relationship between the voltage at which a composite region is executed and the resulting probability of error occurrences during that execution. Based on this information the compiler and/or runtime system can select a mode of operation that depends on the error-handling capabilities specified for this region, and on the environment in which it is executed. For example, in the presence of an elaborate error-recovery capability associated with a function, the system may decide to execute the function at a low voltage, saving energy but accepting the risk of a high number of error occurrences. On the other hand, if an inadequate error response has been provided for the function, the system may choose to executing it at a voltage that minimizes the risk of voltage-induced error.

In general, a voltage clause has the form

voltage_options (v, probability_specification-list)

where v identifies a voltage variable, and the *probability_specification-list* may contain the following elements:

1. An *undetected error probability specification* of the form:

   p_undetected_errors =$p_u(v)$

   where $p_u(v)$ is an expression specifying the probability for undetected errors resulting from an execution at voltage v.

2. A *detected error probability specification* of the form:

   p_detected_errors =$p_d(v)$

   where $p_d(v)$ is an expression specifying the probability for detected errors resulting from an execution at voltage v.

3. An *any error probability specification* of the form:

   p_errors =$p_e(v)$

   where $p_e(v)$ is an expression specifying the probability for any errors resulting from an execution at voltage v.

The probability specification list may consist of exactly one of the above three options, or of a combination of the first two options.

6.4 Examples for Voltage Pragmas

Voltage Pragmas: Example 1

float function f(**float** x,...)
voltage_options (v, **p_undetected_errors** =p1(v),
p_detected_errors =p2(v));
error ($c_1 \rightarrow h_1(...); ..., c_n \rightarrow h_n(...)$);
{...};

In this example, the pragma **voltage_options** (...) is contextually associated with the declaration of function f. The probabilities for undetected as well as detected errors are respectively specified as $p1(v)$ and $p2(v)$ depending on the voltage, v, chosen for an execution of the function. Furthermore, error handlers $h_1, ...h_n$ for error classes $c_1, ...c_n$ are associated with the function.

Voltage Pragmas: Example 2

Here we consider a situation where a function is embedded in a hierarchical program structure and may be executed at different voltages depending on circumstances in its environment. Function arguments in this example are respectively characterized as *input*, *input/output*, or *output* by the qualifiers **in** , **inout** , and **out** .

float function g(**float** **in** x,...)
voltage_options (v, **p_undetected_errors** =p3(v),
p_detected_errors =p4(v));
{...};

Function g has a floating point input argument, x, and delivers a value of type **float** . The voltage pragma provided for the function specifies respective error probabilities $p3(v)$ and $p4(v)$ for undetected and detected errors, depending on the operating voltage at which an invocation of g is executed.

Assume now that $g1$ is a function with input/output argument x that calls g, but does not have any error-handling capabilities associated with its definition:

function g1(**float** **inout** x) {x=g(x)};
...
g1(A);
...

The invocation $g1(A)$ leads to a call of $g(A)$, with its result assigned to variable A if the execution is successful. However, the value of A may be corrupted in the case of an error occurring during the execution of $g(A)$, possibly in addition to the corruption of other data. A compiler or runtime analysis of the environment in which $g1(A)$ is called may determine that such an error is unrecoverable. In this case, the system may decide to execute this call at a high voltage to meet reliability requirements.

On the other hand let function $g2$ be defined as follows:

function g2(**float** **out** x, **in** y) **error** (h1(...)) **repeat** (k, **finally** (h2(...))) **save** (B,C,D);
{x=g(y)};
...
t=A;
g2(A,t);
...

In contrast to $g1$ above, the definition of function $g2$ with output argument x and input argument y has associated with it an explicit error handling specification: in the occurrence of an error, error handler $h1(...)$ is called first, and the function is executed repeatedly up to k times if $h1(...)$ cannot successfully handle the error. If, after k repeated executions, errors still occur, error handler $h2(...)$ is activated. The **save** directive directs the system to save the values of variables B, C, D before the initial invocation of $g2(A, t)$; the original values of these variables are restored before each repeated execution caused by an error occurrence.

The system may at the beginning choose to execute $g2$ at a low voltage, but increase the voltage for repeated executions. However, lowering the voltage initially too far could cause so many retries that the total power consumption actually increases. This could lead to a system strategy for optimizing the voltage initially chosen for minimum energy consumption.

7 Conclusion

We conclude this document by discussing a number of ideas on extending the present specification of the assertion language in Section 7.1, followed by addressing some inherent semantic limitations of the language in Section 7.2.

7.1 Potential Future Extensions of the Assertion Language

There are some obvious areas where the present AL specification can be refined and/or generalized without affecting its basic structure and concepts. They include:

- Dynamic management of assertion language constructs
- Extension of tolerance directives by *approximate types*
- Extended support for error recovery
- Introduction of additional pragma categories
- Definition and management of user-defined error classes
- More precise classifications of data and program regions

It is expected that such changes will be implemented as a result of receiving feedback from compilation of the language and from its application to critical application. The first two of these potential extensions will be respectively discussed in Sections 7.1.1 and 7.1.2. A much further-reaching extension—the introduction of support for parallel computations—will be outlined in Section 7.1.3.

7.1.1 Dynamic Management of Assertion Language Constructs

The discussion in this document assumed that an assertion language construct appearing in a source program was always active. However, compiler and/or runtime analysis may result in the desirability to dynamically manage such constructs, e.g., by turning them off and on depending on the results of analysis. *Knobs*, as proposed by Bernholdt et al. generalize such boolean guards [2]. They are mechanisms that can control the cost, in terms of performance and energy, associated with the execution of assertion language constructs. For example, they allow flexible control of the strength of error detectors such as *checksum* procedures and the frequency of their activation.

7.1.2 Approximate Types

The semantics of most common programming languages specifies in detail the mathematical properties to be obeyed by the representation and processing of numerical types. Tolerance directives, as defined in Section 5 of this document, support some relaxation of these strict rules. A different approach for the manipulation of numerical values could follow the concept of *approximate types* as proposed in the EnerJ language [11]. Approximate types provide flexibility for value representation and operation implementation, with implementation details remaining system and application dependent. The underlying idea is to suppress certain numerical errors, in addition to providing the implementation with the opportunity for achieving potentially higher performance and reduced energy consumption by exploiting the weakened requirements for representations and operations. The introduction of approximate types leads to a separation of computations that must be executed in a mathematically precise fashion from approximate computations. If approximate type computations are mixed with precise type computations, care must be taken not to destroy the hard correctness of precise type computations. Specifically, the assignment of the value of an approximate type variable to a precise-type variable needs special consideration.

7.1.3 Assertion Language Support for Parallelism

The current AL specification is based upon a sequential programming model. In view of the massive parallelism characterizing exascale computations the scope of AL's applicability could be significantly enhanced by introducing support for parallel computations. For example, new AL constructs could support fault tolerance for synchronous and asynchronous parallelism such as forall loops, parallel thread scheduling, and associated synchronization features. Error control would have to be generalized to deal with errors in these new constructs, and at the same time include parallelism in its recovery routines. Predicates could take into account the distribution of large data structures across different memories, addressing related consistency issues and the resulting balance of computations. Tolerance directives could address specific errors in parallel computations that could be ignored without affecting the (soft) correctness of computations. And pragmas could be generalized to control the degree of parallelism of a construct depending on the tradeoff between gain in computational performance and the overhead of synchronization and communication. Such generalizations would result in a major generalization of AL, which we believe could be made without a breakdown of the basic structure of AL as specified at this time.

7.2 On the Semantic Limits of the Assertion Language and Its Implementation

Assertions provide the means to formulate an error detector along the lines of Huang-Abraham's Algorithm-Based Fault Tolerance (ABFT) approach [5]. For example, a call to a *checksum* computation can be ex-

pressed with a status predicate attached to a single-point region. However, AL is not designed to handle error detection capabilities that exploit deep semantic properties of the algorithm and its data structures. Such features must be explicitly included by the programmer in the algorithm and the related data structure declarations.

We illustrate these relationships with an example for a code section, which performs a fault-tolerant computation of the product $y = A * x$ for matrix A and vector x:

/* Original data structures are dimensioned $A(1:m, 1:n), x(1:n), y(1:m)$ */
/* Declaration of extended data structures: */
float AA(1:m+1,1:n); /* Row $AA(m+1, 1:n)$ is added to the original data structure */
float xx(1:n+1); /* Element $xx(n+1)$ is added to the original input vector */
float yy(1:m+1); /* Element $yy(m+1)$ is added to the original result vector */
...
/* Assume $AA(i,j)$ to be defined for $1 \leq i \leq m,\ 1 \leq j \leq n$, and $xx(j)$ for $1 \leq j \leq n$ */
/* $AA(m+1, j)$ is now computed as $AA(m+1, j) = \Sigma_{i=1:m}(A(i,j))$ for all $j,\ 1 \leq j \leq n$ */
/* $xx(n+1)$ is computed as $xx(n+1) = \Sigma_{j=1:n}(x(j))$ */
...
/* compute the matrix-vector product: $yy(1:m) := AA(1:m, 1:n) * xx(1:n)$ */
/* compute $yy(m+1) := AA(m+1) * xx(1:n)$ */
assert (**post** $(yy(m+1) = \Sigma_{i=1:m}(yy(i)))$) **error** (checksum_error_handler(AA, xx, yy));
...

In the case that the postcondition for the last statement fails, the *checksum_error_handler(...)* will be activated and given the task to perform recovery from that error. However, we do not assume that the system in which the implementation of AL is embedded has sufficient knowledge about the algorithmic and data structure semantics to deal with this problem successfully on its own. That is, any information required for the analysis of and recovery from such an error must be explicitly provided in the error handler.

References

[1] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. Fundamental Concepts of Dependability. Technical report, UCLA, 2000. CSD Report No. 010028.

[2] David E. Bernholdt, Wael R. Elwasif, Christos Kartsaklis, Seyong Lee, and Tiffany M. Mintz. Programmer-Guided Reliability and Trade-Offs with Energy and Performance. Slides presented at ACI PI Meeting, Baltimore, MD, February 2014.

[3] Patrick G. Bridges, Kurt B. Ferrera, Michael A. Heroux, and Mark Hoemmen. Fault-Tolerant Iterative Methods via Selective Reliability. Technical report, Sandia national laboratories, June 2012.

[4] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[5] K.H. Huang and J.A. Abraham. Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Transactions on Computers*, C-33:518–528, June 1984.

[6] Mark L. James and Lydia P. Dubon. An Autonomous Diagnostic and Prognostic Monitoring System for NASA's Deep Space Network. In *Proceedings 2001 IEEE Aerospace Conference*, March 2001.

[7] Xuanhua Li and Donald Yeung. Application-level correctness and its impact on fault tolerance. In *In Proceedings of the 13th International Symposium on High Performance Computer Architecture*, pages 181–192, 2007.

[8] Bertrand Meyer. Design by Contract. In *Advances in Object-Oriented Software Engineering*, pages 1–50. Prentice Hall, 1991.

[9] Microsoft. Code Contracts. http://research.microsoft.com/en-us/projects/contracts.

[10] Oracle Software Downloads. Programming With Assertions. http://docs.oracle.com/javase/1.4.2/docs/guide/lang/assert.html, 2002.

[11] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 164–174, New York, NY, USA, 2011. ACM.

[12] Lotfi A. Zadeh. Fuzzy logic, neural networks, and soft computing. *Commun. ACM*, 37(3):77–84, March 1994.

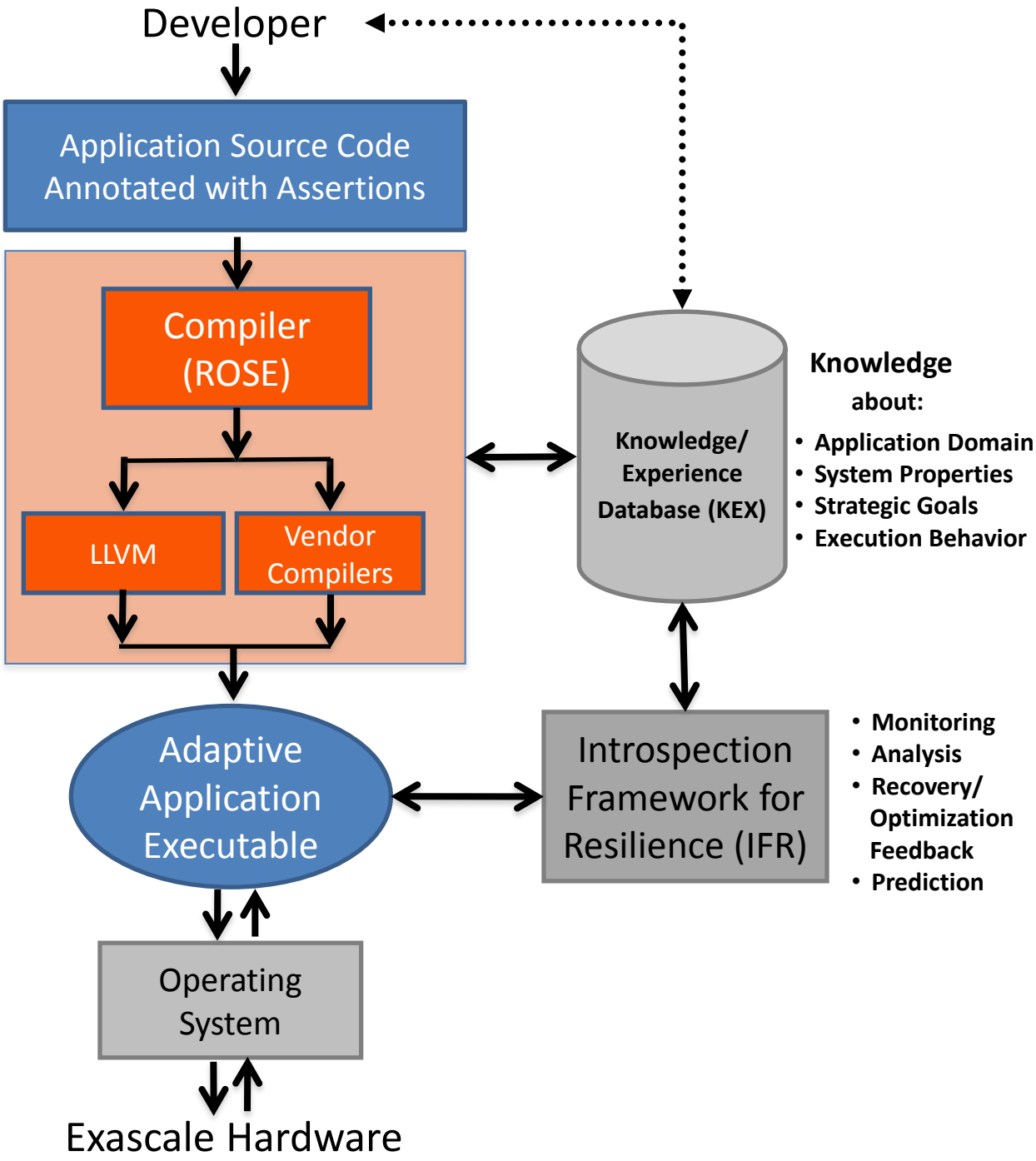

Figure 1: FailSafe System Overview